Interface-Driven, Model-Based Test Automation

Dr. Mark R. Blackburn, Robert D. Busser, and Aaron M. Nauman
Software Productivity Consortium

Thursday, 1 May 2003
Track 8: 9:15 - 9:55
Room 251 D - F

This article describes an interface-driven approach that combines requirements modeling to support automated test-case and test-driver generation. It focuses on how test engineers can develop more reusable models by clarifying textual requirements as models in terms of component or system interfaces. The focus of interface-driven modeling has been recognized as an improvement over the process of requirements-driven, model-based test automation. Model-based testing users identified the insights and recommendations for applying this approach when they began to scale their model-based testing efforts on larger projects.

Model-based test automation has helped reduce cost, provide early identification of requirements defects, and improve test coverage [1, 2, 3, 4, 5, 6]. Industry use of model-based test automation has provided insight into practical methods that use interface-driven analysis with requirements modeling to support automated test generation. However, the term interface is used loosely in this article.

An *interface* is a component's inputs and outputs, along with the mechanism to set inputs, including state and history information, and retrieve the resulting outputs. Recommendations are provided to perform the modeling of textual requirements in conjunction with interface analysis to support reuse of models and their associated test driver mappings.

Test-driver mappings specify the relationships between model variables and the interfaces of the system under test. The insights are useful for understanding how to scale models and the associated test-driver mappings to support industry-sized verification projects, while supporting organizational integration that helps leverage key resources.

We have applied the model-based test-automation method called the Test Automation Framework (TAF) since 1996. TAF integrates various government and commercially available model-development and test-generation tools to support defect prevention and automated testing of systems and software. TAF supports modeling methods that focus on representing requirements such as the Software Cost Reduction (SCR) method as well as methods that focus on representing design information such as Unified Modeling Language (UML)-based tools or Mathwork's Simulink, which supports control system modeling for automotive and aircraft systems.

With model translation, requirements-based or design-based models are converted into test specifications. T-VEC is the test generation component of TAF that uses the test specification to produce tests. T-VEC supports test-vector generation, test-driver generation, requirements test-coverage analysis, and test results checking and reporting. *Test vectors* include inputs as well as the expected outputs with requirement-to-test traceability information. The test-driver mappings and the test vectors are inputs to the test-driver generator that produces test drivers. The test drivers are then executed against the implemented system during test execution.

There are papers that describe requirements modeling [7] and others with examples that support automated test generation [3, 8, 9, 10]. Aissi provides a historical perspective on test-vector generation and describes some of the leading commercial tools [11]. Pretschner and Lotzbeyer briefly discuss extreme modeling that includes model-based test generation [12], which is similar to uses of TAF. There are various approaches to model-based testing and Robinson hosts a Web site that provides useful links to authors, tools, and papers [13].

Why Interface-Driven Modeling?

It may seem appropriate first to develop models from the requirements, but when developing models for testing purposes, the models should be developed in conjunction with analysis of the interfaces to the component or system under test. Modeling the behavioral requirements is usually straightforward and easier to evolve once the interfaces and operations are understood because the behavioral requirements, usually defined in text, must be modeled in terms of variables that represent objects accessible through interfaces.

A *verification model* is a refinement of the requirements specified in terms of the component's interfaces. Verification modeling from the interfaces is analogous to a test engineer developing tests in terms of specific interfaces of the component under test. The test engineer's role involves developing verification models from requirements. The requirements engineers, sometimes synonymous with system engineers, continue to develop textual requirements as well as any other type of analytical model. Design engineers should focus on identifying the components of the system architecture, and provide the component interfaces to the test engineer.

These requirements and interfaces can then be used by test engineers to construct formalized verification models (in a tool like SCR). The test engineer can use TAF tools to perform model analysis and correct any inconsistency, as well as produce test vectors and test drivers. Once completed, the textual requirements and models, as well as the formalized verification models are passed to the designers and implementers.

Once code is created, the generated test drivers can be used to test the implementation. TAF translators convert verification models into a form where the T-VEC system generates test vectors and test drivers, with requirement-to-test traceability information that allows failures to be traced backwards to the requirement. The designer/implementer can then use the test drivers to test the implemented system, continuously and iteratively.

Figure 1 on page 28 provides a perspective of the verification modeling process flow. A modeler is supplied with various inputs. Although it is common to start the process with poorly defined requirements, inputs to the process can include requirement system and software specifications (e.g., System Requirements Specification [SRS], Software Requirements Specifications [SWRS]), user documentation, interface control documents, application program interface documents, previous designs, and old test scripts.

A verification model is composed of a model and one or more test-driver mappings. A test driver consists of object mappings and a schema (pattern). Object mappings relate the model objects to the interfaces of the system under test. The schema defines the algorithmic pattern to carry out the execution of the test cases. Models are typically

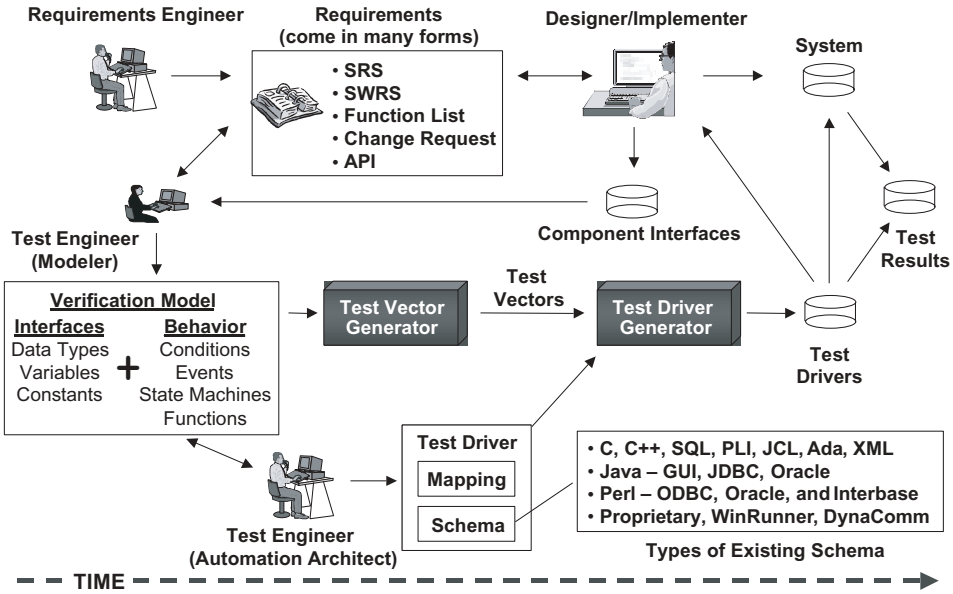


Figure 1: *Interface-Driven, Model-Based Test Automation Supports Continuous Verification and Validation*

developed incrementally. The test-vector generator also detects non-testable modeled requirements (i.e., requirements with contradictions).

Table-based modeling, like the SCR method, has been effective and relatively easy to learn for test engineers [2]. Design engineers commonly develop models based on state machines or other notations such as the UML. However, users and project leaders observed that test engineers found it easier to develop requirements for test in the form of tables (See [8] for details). The modeling notations supported by tools for the SCR method have well-defined syntax and semantics allowing for a precise and analyzable definition of the required behavior.

What Are Some Modeling Perspectives?

Specification languages, usually supported through graphical modeling environments, describe models. Specification languages provide abstract descriptions of system and software requirements and design information. Cooke et al. developed a scheme that classified specification language characteristics [14]. Independent of any specification language, Figure 2 illustrates three specification categories based on the purpose of the specification. Cooke et al. indicate that most specification languages usually are based on a hybrid approach that integrates different classes of specifications.

Requirements specifications define the boundaries between the environment and the system and, as a result, impose constraints on the system. Functional specifications define behavior in terms of the interfaces between components, and design specifications define the component itself. A verification model, in the context of this article, is best classified as a functional specification.

Design-for-Test Supports Interface Accessibility

It is best to understand the interfaces of the system under test before modeling the behavioral requirements to ensure that the interfaces for the resulting test-driver map to actual inputs or outputs of the system under test. If the interfaces are not formalized or completely understood, requirements models can be developed, but associated object mappings required to support test-driver generation must be completed after the interfaces have been formalized.

This can make the object-mapping process more complex because the model entities may not map to the component interfaces. In addition, if the component interfaces are coupled to other components, the components are typically not completely controllable through separate interfaces. This too can complicate the modeling and testing process. Consider the following conceptual representation of the set of components and interfaces shown in Figure 3.

There is a specific way to support a systematic verification approach that can be performed in stages where each component is completely verified with respect to the requirements allocated to it. The interfaces to the component should be explicitly and completely accessible, either using global memory or, better, through get-and-set methods/procedures in Figure 3.

For example, if the inputs to the B.2 component of higher-level component B are completely available for setting the inputs to B.2, and the outputs from the B.2 functions can be completely observed, then the functionality within B.2 can be completely specified and systematically verified. However, if interfaces from other components such as B.1 are not accessible then some of the functionality of the B.2 component is coupled with B.1, and the interfaces to B.2 must also include interfaces to B.1 or to other upstream components such as component A. This interface coupling makes the test-driver interfaces more complex to describe, but also forces the behavioral modeling to be described in terms of functionality allocated to combinations of components.

The coupling reduces components' reuse and increases the regression testing effort due to the coupled aspects of the system components. The problems associated with testing highly coupled systems can be problematic for model-based testing and also negatively affect any type of testing. We have observed that interface-driven modeling has helped foster better system design by reducing the coupling, but also has helped provide better testing support.

Understanding Interfaces

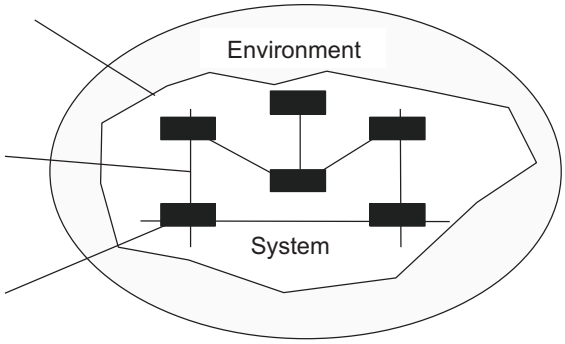
One of the most noted results of the TAF was its application to the Mars Polar Lander (MPL). NASA launched the MPL project on Feb. 7, 1994. Six years later on Dec. 3, 1999, after the MPL had traveled more than 35

Figure 2: *Specification Purposes*

Requirements Specification: Defines the boundary between the environment and the system.

Functional Specification: Defines the interfaces within the system.

Design Specification: Defines the component.



Note: D. Cooke et al., 1996

million miles and was minutes away from its scheduled landing, all contact with the craft was lost. The MPL cost $165 million to develop and deploy.

After the fact, we had the opportunity to use the TAF to see if it would have found the bug. We deliberately did not look at the code before creating our tests. Instead, we created them by modeling the English-language requirements in a tool based on the SCR method. We modeled the Touchdown Monitor (TDM) requirements using the TAF tools and were able to identify the software error associated with the MPL's landing procedures in fewer than 24 hours.

The TDM is a software component of the MPL system that monitors the state of three landing legs during two stages of the descent. As shown in Figure 4, a real-time multi-tasking executive calls the TDM module at a rate of 100 times per second and receives information on the leg sensors from a second module. These two modules establish the interfaces to TDM. During the first stage, starting approximately five kilometers above the Mars surface, the TDM software monitors the three touchdown legs.

There is one sensor on each leg to determine if the leg touched down. When the legs lock into the deployed position, there was a known possibility that the sensor might indicate an erroneous touchdown signal. The TDM software was to handle this potential event by marking a leg that generates a spurious signal on two consecutive sensor-reads as having a bad sensor. During the second stage, starting about 40 meters above the surface, the TDM software was to monitor the remaining *good* sensors. When a sensor had two consecutive reads indicating touchdown, the TDM software was to command the descent engine to shutdown.

There is no absolute way to confirm what happened to the MPL. It is believed that one or more of the sensors did have two consecutive reads before the 40 meter point; the leg-sensor information was stored in TDM program memory. When the MPL crossed the 40-meter point, the TDM changed states and read the memory associated with the leg-sensors during the first stage of descent. Because the memory indicated two consecutive reads engineers believe that the engine thruster was shut off at about 40 meters above the Mars surface. Developers could have designed and implemented the requirement in many ways, but the essence of the design flaw is that the program variables retained the state of the bad sensor information.

Organizational Best Practices

Interface-driven modeling can be applied after development is complete; however, significant benefits have been realized when it is applied during development. Ideally, test engineers work in parallel with developers to stabilize interfaces, refine requirements, and build models to support iterative test and development. Test engineers write the requirements for the products (which in some cases are poorly documented) in the form of models, as opposed to hundreds or thousands of lines of test scripts. They generate the test vectors and test drivers automatically.

During iterative development, if the component behavior, the interface, or the requirements change, the models are modified and test cases and test drivers are regenerated and re-executed. The key advantages are that testing proceeds in parallel with development. Users like Lockheed Martin state that test is being reduced by about 50 percent or more, while describing how early requirement analysis significantly reduces rework through elimination of requirement defects (i.e., contradiction, inconsistencies, and feature interaction problems) [2, 15].

Other Applications

TAF has been applied to applications in various domains, including critical applications for aerospace, medical devices, flight navigation, guidance, autopilots, display systems, flight management and control laws, engine controls, and airborne traffic and collision avoidance. TAF has also been applied to noncritical applications like databases, client-server, Web-based, automotive, and telecommunication applications. The related test-driver generation has been developed for many languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL, etc.) as well as proprietary languages, COTS test-injection products (e.g., DynaComm, WinRunner) and test environments. Most users of the approach have reduced their verification/test effort by 50 percent [2, 15].

Summary

This article provides pragmatic guidance for combining interface analysis and requirements modeling to support model-based test automation. The model-based testing method and tools described here have been demonstrated to significantly reduce cost and effort for performing testing, while also being demonstrated to identify requirements defects that reduce costly rework.

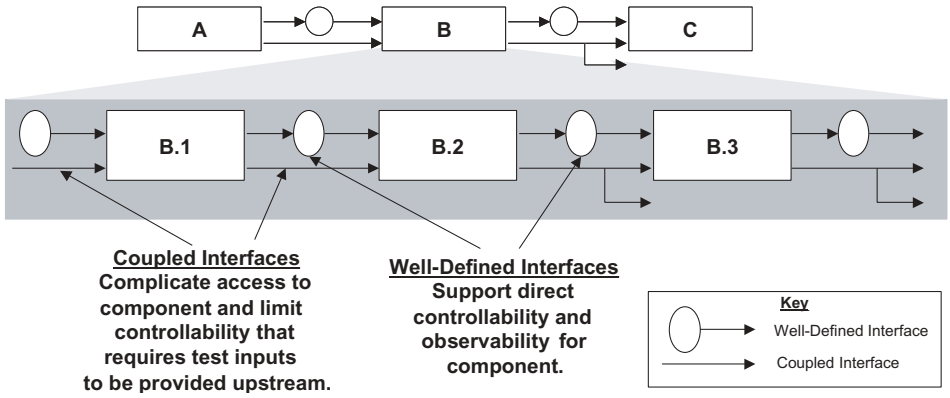


Figure 3: *Conceptual Components and Interfaces of a System*

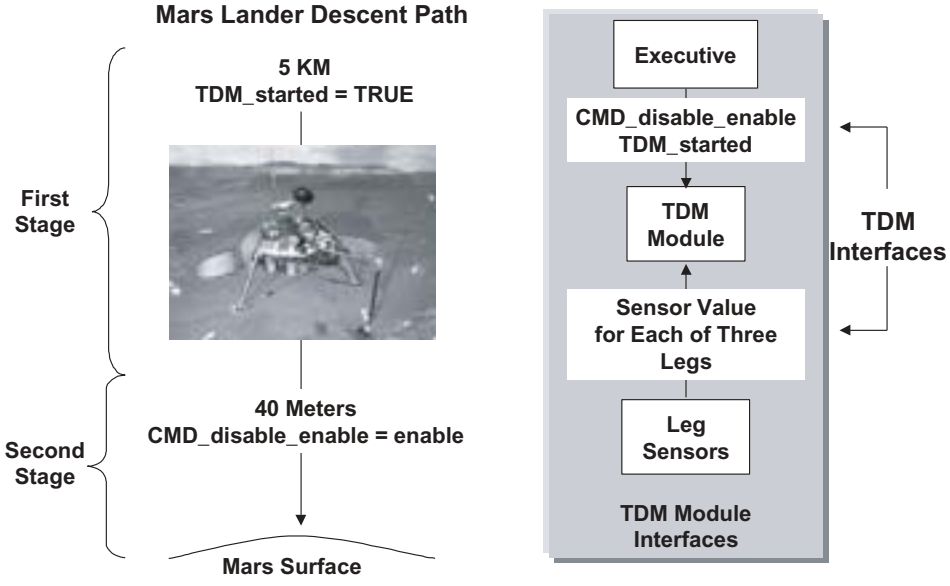


Figure 4: *Mars Polar Lander Details*

These recommendations for defining interfaces that provide better support for testability are valid for all forms of testing. Organizations can see the benefits of using interface-driven, model-based testing to help stabilize the interfaces of the system early, while identifying common test-driver support capabilities that can be constructed once and reused across related tests. Finally, parallel development of verification modeling is beneficial in development and helps identify requirement defects early to reduce rework.

The TAF is a framework that integrates various commercial and government tools, however, the Software Productivity Consortium does not provide licenses for these tools. For additional information on obtaining these tools, contact the authors. Finally, many referenced papers in the article can be downloaded from the Software Productivity Consortium Web site <www.software.org/pub/taf/Reports.html>.◆

References

1. Rosario, S., and H. Robinson. "Applying Models in Your Testing Process, Information and Software Technology." 42.12 (1 Sept. 2000).
2. Kelly, V., E. L. Safford, M. Siok, and M. Blackburn. "Requirements Testability and Test Automation," Lockheed Martin Joint Symposium, June 2001.
3. Blackburn, M. R., R. D. Busser, A. M. Nauman, R. Knickerbocker, and R. Kasuda. "Mars Polar Lander Fault Identification Using Model-Based Testing." Proc. in IEEE/NASA 26th Software Engineering Workshop, Nov. 2001.
4. Busser, R. D., M. R. Blackburn, and A. M. Nauman. "Automated Model Analysis and Test Generation for Flight Guidance Mode Logic." Digital Avionics System Conference. Indianapolis, IN, 2001.
5. Statezni, David. "Test Automation Framework, State-Based, and Signal Flow Examples." Twelfth Annual Software Technology Conference. Salt Lake City, UT, 30 Apr.-5 May 2000.
6. Statezni, David. "T-VEC's Test Vector Generation System." Software Testing and Quality Engineering. May/June 2001.
7. Heitmeyer, C., R. Jeffords, and B. Labaw. "Automated Consistency Checking of Requirements Specifications." ACM TOSEM 5.3 (1996): 231-261.
8. Blackburn, M. R., R. D. Busser, and A. M. Nauman. "Removing Requirement Defects and Automating Test." STAR-EAST. Orlando, FL, May 2001.
9. Blackburn, M. R., R. D. Busser, and A. M. Nauman. "Eliminating Requirement Defects and Automating Test." Test Computer Software Conference, June 2001.
10. Blackburn, M. R., R. D. Busser, A. M. Nauman, and R. Chandramouli. Model-Based Approach to Security Test Automation. Proc. of Quality Week, June 2001.
11. Aissi, S. "Test Vector Generation: Current Status and Future Trends." Software Quality Professional 4.2 (Mar. 2002).
12. Pretschner, A., and H. Lotzbeyer. Model-Based Testing with Constraint Logic Programming: First Results and Challenges. Proc. of 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification. Toronto, Canada, May 2001.
13. Robinson, H. Model-Based Testing <www.model-based-testing.org>.
14. Cooke, D., A. Gates, E. Demirors, O. Demirors, M. Tankik, and B. Kramer. "Languages for the Specification of Software." Journal of Systems Software 32 (1996): 269-308.21.
15. Safford, Ed L. "Test Automation Framework, State-Based and Signal Flow Examples." Twelfth Annual Software Technology Conference. Salt Lake City, UT, 30 Apr.-5 May 2000.

About the Authors

Mark R. Blackburn, Ph.D., is a Software Productivity Consortium fellow with 20 years of software systems engineering experience in development, management and applied research of process, methods, and tools. He has made more than 30 presentations at conferences and symposia, and is involved in consulting, strategic planning, proposal and business development, as well as developing and applying methods for model-based approaches for requirement defect removal and test automation.

Software Productivity Consortium
2214 Rock Hill Road
Herndon,VA 20170
Phone: (703) 742-7136
Fax: (703) 742-7350
E-mail: blackbur@software.org

Robert D. Busser is a principal member of the technical staff of the Software Productivity Consortium. He has more than 20 years of software systems engineering experience in development, and management in the area of advanced software engineering, and expertise in software engineering processes, methods and tools. He has extensive experience in requirement and design methods, real-time systems, model-based development and test generation tools, model analysis, and verification.

Software Productivity Consortium
2214 Rock Hill Road
Herndon,VA 20170
Phone: (954) 753-9634
Fax: (703) 742-7350
E-mail: busser@software.org

Aaron M. Nauman is a senior member of the technical staff of the Software Productivity Consortium. He has a wide range of systems and applications development experience in both real-time and information systems domains. Nauman is involved in the development of model-based transformation tools for automated model analysis and test generation. He has experience in object-oriented technologies, distributed and client/server systems, Web-based and components-based software and systems integration.

Software Productivity Consortium
2214 Rock Hill Road
Herndon,VA 20170
Phone: (703) 742-7104
Fax: (703) 742-7350
E-mail: nauman@software.org